

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

2

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS
BEFORE COMPLETING FORM

1. REPORT NUMBER DTIC FILE COPY		12. GOVT ACCESSION NO.	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) Ada Compiler Validation Summary Report: Rockwell International Corporation, DDC-Based Ada/CAPS Compiler, Version 4.1, VAXstation 3100 Model 30 (Host) to CAPS/AAMP2 (Target), ACVC 1.10.		5. TYPE OF REPORT & PERIOD COVERED 01 Nov. 1989 to 01 Dec. 1990	
7. AUTHOR(s) Wright-Patterson AFB Dayton, OH, USA		6. PERFORMING ORG. REPORT NUMBER	
9. PERFORMING ORGANIZATION AND ADDRESS Wright-Patterson AFB Dayton, OH, USA		8. CONTRACT OR GRANT NUMBER(s)	
11. CONTROLLING OFFICE NAME AND ADDRESS Ada Joint Program Office United States Department of Defense Washington, DC 20301-3081		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS	
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office) Wright-Patterson AFB Dayton, OH, USA		12. REPORT DATE	
		13. NUMBER OF PAGES	
		15. SECURITY CLASS (of this report) UNCLASSIFIED	
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE N/A	
16. DISTRIBUTION STATEMENT (of this Report) Approved for public release; distribution unlimited.			
17. DISTRIBUTION STATEMENT (of the abstract entered on page 20, if different from Report) UNCLASSIFIED			
18. SUPPLEMENTARY NOTES			
19. KEYWORDS (Continue on reverse side if necessary and identify by block number) Ada Programming language, Ada Compiler Validation Summary Report, Ada Compiler Validation Capability, ACVC, Validation Testing, Ada Validation Office, AVO, Ada Validation Facility, AVF, ANSI/MIL-STD-1815A, Ada Joint Program Office, AJPO			
20. ABSTRACT (Continue on reverse side if necessary and identify by block number) Rockwell International Corporation DDC-Based Ada/CAPS Compiler, Version 4.1, Wright-Patterson AFB, OH, VAXstation 3100 Model 30 under VMS, 5.2 (Host) to CAPS/AAMP2 under bare machine (Target), ACVC 1.10.			

DTIC
ELECTE
MAR 15 1990
S D C D

DD FORM 1473
1 JAN 73EDITION OF 1 NOV 65 IS OBSOLETE
S/N 0102-LF-014-8601

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

AD-A219 052

AVF Control Number: AVF-VSR-333.0190
89-07-21-RWL

Ada COMPILER
VALIDATION SUMMARY REPORT:
Certificate Number: 891101W1.10182
Rockwell International Corporation
DDC-Based Ada/CAPS Compiler, Version 4.1
VAXstation 3100 Model 30 Host and CAPS/AAMP2 Target

Completion of On-Site Testing:
01 November 1989

Prepared By:
Ada Validation Facility
ASD/SCEL
Wright-Patterson AFB OH 45433-6503

Prepared For:
Ada Joint Program Office
United States Department of Defense
Washington DC 20301-3081

90-03-14 020

Ada Compiler Validation Summary Report:

Compiler Name: DDC-Based Ada/CAPS Compiler, Version 4.1

Certificate Number: 891101W1.10182


Host: VAXstation 3100 Model 30 under
VMS, 5.2

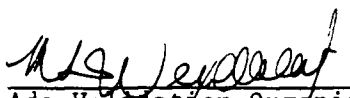
Target: CAPS/AAMP2 under
bare machine


Testing Completed 01 November 1989 Using ACVC 1.10

Customer Agreement Number: 89-07-21-RWL

This report has been reviewed and is approved.


Ada Validation Facility
Steven P. Wilson
Technical Director
ASD/SCEL
Wright-Patterson AFB OH 45433-6503


Ada Validation Organization
Dr. John F. Kramer
Institute for Defense Analyses
Alexandria VA 22311


Ada Joint Program Office
Dr. John Solomond
Director
Department of Defense
Washington DC 20301

Accession For	
NTIS	<input checked="checked" type="checkbox"/>
DTIC	<input type="checkbox"/>
DDP	<input type="checkbox"/>
By	
Date	
Activity Codes	
Dist	Adm. or Special
A-1	



TABLE OF CONTENTS

CHAPTER 1	INTRODUCTION	
1.1	PURPOSE OF THIS VALIDATION SUMMARY REPORT	1-2
1.2	USE OF THIS VALIDATION SUMMARY REPORT	1-2
1.3	REFERENCES.	1-3
1.4	DEFINITION OF TERMS	1-3
1.5	ACVC TEST CLASSES	1-4
CHAPTER 2	CONFIGURATION INFORMATION	
2.1	CONFIGURATION TESTED.	2-1
2.2	IMPLEMENTATION CHARACTERISTICS.	2 2
CHAPTER 3	TEST INFORMATION	
3.1	TEST RESULTS.	3-1
3.2	SUMMARY OF TEST RESULTS BY CLASS.	3-1
3.3	SUMMARY OF TEST RESULTS BY CHAPTER.	3-2
3.4	WITHDRAWN TESTS	3-2
3.5	INAPPLICABLE TESTS.	3-2
3.6	TEST, PROCESSING, AND EVALUATION MODIFICATIONS. .	3-6
3.7	ADDITIONAL TESTING INFORMATION.	3-6
3.7.1	Prevalidation	3-6
3.7.2	Test Method	3-6
3.7.3	Test Site	3-8
APPENDIX A	DECLARATION OF CONFORMANCE	
APPENDIX B	APPENDIX F OF THE Ada STANDARD	
APPENDIX C	TEST PARAMETERS	
APPENDIX D	WITHDRAWN TESTS	
APPENDIX E	COMPILER OPTIONS AS SUPPLIED BY ROCKWELL INTERNATIONAL CORPORATION	

CHAPTER 1

INTRODUCTION

This Validation Summary Report (VSR) describes the extent to which a specific Ada compiler conforms to the Ada Standard, ANSI/MIL-STD-1815A. This report explains all technical terms used within it and thoroughly reports the results of testing this compiler using the Ada Compiler Validation Capability (ACVC). An Ada compiler must be implemented according to the Ada Standard, and any implementation-dependent features must conform to the requirements of the Ada Standard. The Ada Standard must be implemented in its entirety, and nothing can be implemented that is not in the Standard.

Even though all validated Ada compilers conform to the Ada Standard, it must be understood that some differences do exist between implementations. The Ada Standard permits some implementation dependencies--for example, the maximum length of identifiers or the maximum values of integer types. Other differences between compilers result from the characteristics of particular operating systems, hardware, or implementation strategies. All the dependencies observed during the process of testing this compiler are given in this report.

The information in this report is derived from the test results produced during validation testing. The validation process includes submitting a suite of standardized tests, the ACVC, as inputs to an Ada compiler and evaluating the results. The purpose of validating is to ensure conformity of the compiler to the Ada Standard by testing that the compiler properly implements legal language constructs and that it identifies and rejects illegal language constructs. The testing also identifies behavior that is implementation-dependent but is permitted by the Ada Standard. Six classes of tests are used. These tests are designed to perform checks at compile time, at link time, and during execution.

INTRODUCTION

1.1 PURPOSE OF THIS VALIDATION SUMMARY REPORT

This VSR documents the results of the validation testing performed on an Ada compiler. Testing was carried out for the following purposes:

- . To attempt to identify any language constructs supported by the compiler that do not conform to the Ada Standard
- . To attempt to identify any language constructs not supported by the compiler but required by the Ada Standard
- . To determine that the implementation-dependent behavior is allowed by the Ada Standard

Testing of this compiler was conducted by SofTech, Inc. under the direction of the AVF according to procedures established by the Ada Joint Program Office and administered by the Ada Validation Organization (AVO). On-site testing was completed 01 November 1989 at Cedar Rapids IA

1.2 USE OF THIS VALIDATION SUMMARY REPORT

Consistent with the national laws of the originating country, the AVO may make full and free public disclosure of this report. In the United States, this is provided in accordance with the "Freedom of Information Act" (5 U.S.C.#552). The results of this validation apply only to the computers, operating systems, and compiler versions identified in this report.

The organizations represented on the signature page of this report do not represent or warrant that all statements set forth in this report are accurate and complete, or that the subject compiler has no nonconformities to the Ada Standard other than those presented. Copies of this report are available to the public from:

Ada Information Clearinghouse
Ada Joint Program Office
OUSDRE
The Pentagon, Rm 3D-139 (Fern Street)
Washington DC 20301-3081

or from:

Ada Validation Facility
ASD/SCSL
Wright-Patterson AFB OH 45433-6503

INTRODUCTION

Questions regarding this report or the validation test results should be directed to the AVF listed above or to:

Ada Validation Organization
Institute for Defense Analyses
1801 North Beauregard Street
Alexandria VA 22311

1.3 REFERENCES

1. Reference Manual for the Ada Programming Language, ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
2. Ada Compiler Validation Procedures, Version 2.0, Ada Joint Program Office, May 1989.
3. Ada Compiler Validation Capability Implementers' Guide, SofTech, Inc., December 1986.
4. Ada Compiler Validation Capability User's Guide, December 1986.

1.4 DEFINITION OF TERMS

ACVC	The Ada Compiler Validation Capability. The set of Ada programs that tests the conformity of an Ada compiler to the Ada programming language.
Ada Commentary	An Ada Commentary contains all information relevant to the point addressed by a comment on the Ada Standard. These comments are given a unique identification number having the form AI-ddddd.
Ada Standard	ANSI/MIL-STD-1815A, February 1983 and ISO 8652-1987.
Applicant	The agency requesting validation.
AVF	The Ada Validation Facility. The AVF is responsible for conducting compiler validations according to procedures contained in the <u>Ada Compiler Validation Procedures</u> .
AVO	The Ada Validation Organization. The AVO has oversight authority over all AVF practices for the purpose of maintaining a uniform process for validation of Ada compilers. The AVO provides administrative and technical support for Ada validations to ensure consistent practices.
Compiler	A processor for the Ada language. In the context of this report, a compiler is any language processor, including cross-compilers, translators, and interpreters.

INTRODUCTION

Failed test	An ACVC test for which the compiler generates a result that demonstrates nonconformity to the Ada Standard.
Host	The computer on which the compiler resides.
Inapplicable test	An ACVC test that uses features of the language that a compiler is not required to support or may legitimately support in a way other than the one expected by the test.
Passed test	An ACVC test for which a compiler generates the expected result.
Target	The computer for which a compiler generates code.
Test	A program that checks a compiler's conformity regarding a particular feature or a combination of features to the Ada Standard. In the context of this report, the term is used to designate a single test, which may comprise one or more files.
Withdrawn test	An ACVC test found to be incorrect and not used to check conformity to the Ada Standard. A test may be incorrect because it has an invalid test objective, fails to meet its test objective, or contains illegal or erroneous use of the language.

1.5 ACVC TEST CLASSES

Conformity to the Ada Standard is measured using the ACVC. The ACVC contains both legal and illegal Ada programs structured into six test classes: A, B, C, D, E, and L. The first letter of a test name identifies the class to which it belongs. Class A, C, D, and E tests are executable, and special program units are used to report their results during execution. Class B tests are expected to produce compilation errors. Class L tests are expected to produce compilation or link errors because of the way in which a program library is used at link time.

Class A tests ensure the successful compilation of legal Ada programs with certain language constructs which cannot be verified at compile time. There are no explicit program components in a Class A test to check semantics. For example, a Class A test checks that reserved words of another language (other than those already reserved in the Ada language) are not treated as reserved words by an Ada compiler. A Class A test is passed if no errors are detected at compile time and the program executes to produce a PASSED message.

Class B tests check that a compiler detects illegal language usage. Class B tests are not executable. Each test in this class is compiled and the resulting compilation listing is examined to verify that every syntax or semantic error in the test is detected. A Class B test is passed if every illegal construct that it contains is detected by the compiler.

Class C tests check the run time system to ensure that legal Ada programs can be correctly compiled and executed. Each Class C test is self-checking and produces a PASSED, FAILED, or NOT APPLICABLE message indicating the result when it is executed.

Class D tests check the compilation and execution capacities of a compiler. Since there are no capacity requirements placed on a compiler by the Ada Standard for some parameters--for example, the number of identifiers permitted in a compilation or the number of units in a library--a compiler may refuse to compile a Class D test and still be a conforming compiler. Therefore, if a Class D test fails to compile because the capacity of the compiler is exceeded, the test is classified as inapplicable. If a Class D test compiles successfully, it is self-checking and produces a PASSED or FAILED message during execution.

Class E tests are expected to execute successfully and check implementation-dependent options and resolutions of ambiguities in the Ada Standard. Each Class E test is self-checking and produces a NOT APPLICABLE, PASSED, or FAILED message when it is compiled and executed. However, the Ada Standard permits an implementation to reject programs containing some features addressed by Class E tests during compilation. Therefore, a Class E test is passed by a compiler if it is compiled successfully and executes to produce a PASSED message, or if it is rejected by the compiler for an allowable reason.

Class L tests check that incomplete or illegal Ada programs involving multiple, separately compiled units are detected and not allowed to execute. Class L tests are compiled separately and execution is attempted. A Class L test passes if it is rejected at link time--that is, an attempt to execute the main program must generate an error message before any declarations in the main program or any units referenced by the main program are elaborated. In some cases, an implementation may legitimately detect errors during compilation of the test.

Two library units, the package REPORT and the procedure CHECK_FILE, support the self-checking features of the executable tests. The package REPORT provides the mechanism by which executable tests report PASSED, FAILED, or NOT APPLICABLE results. It also provides a set of identity functions used to defeat some compiler optimizations allowed by the Ada Standard that would circumvent a test objective. The procedure CHECK_FILE is used to check the contents of text files written by some of the Class C tests for chapter 14 of the Ada Standard. The operation of REPORT and CHECK_FILE is checked by a set of executable tests. These tests produce messages that are examined to verify that the units are operating correctly. If these units are not operating correctly, then the validation is not attempted.

The text of each test in the ACVC follows conventions that are intended to ensure that the tests are reasonably portable without modification. For example, the tests make use of only the basic set of 55 characters, contain lines with a maximum length of 72 characters, use small numeric values, and place features that may not be supported by all implementations in separate tests. However, some tests contain values that require the test to be

INTRODUCTION

customized according to implementation-specific values--for example, an illegal file name. A list of the values used for this validation is provided in Appendix C.

A compiler must correctly process each of the tests in the suite and demonstrate conformity to the Ada Standard by either meeting the pass criteria given for the test or by showing that the test is inapplicable to the implementation. The applicability of a test to an implementation is considered each time the implementation is validated. A test that is inapplicable for one validation is not necessarily inapplicable for a subsequent validation. Any test that was determined to contain an illegal language construct or an erroneous language construct is withdrawn from the ACVC and, therefore, is not used in testing a compiler. The tests withdrawn at the time of this validation are given in Appendix D.

CHAPTER 2

CONFIGURATION INFORMATION

2.1 CONFIGURATION TESTED

The candidate compilation system for this validation was tested under the following configuration:

Compiler: DDC-Based Ada/CAPS Compiler, Version 4.1

ACVC Version: 1.10

Certificate Number: 891101W1.10182

Host Computer:

Machine: VAXstation 3100 Model 30

Operating System: VMS 5.2

Memory Size: 16MB

Target Computer:

Machine: CAPS/AAMP2

Operating System: bare machine

Memory Size: 192K

Communications Network: Ethernet

CONFIGURATION INFORMATION

2.2 IMPLEMENTATION CHARACTERISTICS

One of the purposes of validating compilers is to determine the behavior of a compiler in those areas of the Ada Standard that permit implementations to differ. Class D and E tests specifically check for such implementation differences. However, tests in other classes also characterize an implementation. The tests demonstrate the following characteristics:

a. Capacities.

- (1) The compiler correctly processes a compilation containing 723 variables in the same declarative part. (See test D29002K.)
- (2) The compiler correctly processes tests containing loop statements nested to 65 levels. (See tests D55A03A..H (8 tests).)
- (3) The compiler correctly processes tests containing block statements nested to 65 levels. (See test D56001B.)
- (4) The compiler correctly processes tests containing recursive procedures separately compiled as subunits nested to 17 levels. (See tests D64005E..G (3 tests).)

b. Predefined types.

- (1) This implementation supports the additional predefined types LONG_INTEGER and LONG_FLOAT in package STANDARD. (See tests B86001T..Z (7 tests).)

c. Expression evaluation.

The order in which expressions are evaluated and the time at which constraints are checked are not defined by the language. While the ACVC tests do not specifically attempt to determine the order of evaluation of expressions, test results indicate the following:

- (1) None of the default initialization expressions for record components are evaluated before any value is checked for membership in a component's subtype. (See test C32117A.)
- (2) Assignments for subtypes are performed with the same precision as the base type. (See test C35712B.)
- (3) This implementation uses no extra bits for extra precision and uses all extra bits for extra range. (See test C35903A.)

CONFIGURATION INFORMATION

- (4) Sometimes `NUMERIC_ERROR` is raised when an integer literal operand in a comparison or membership test is outside the range of the base type. (See test C45232A.)
- (5) `NUMERIC_ERROR` is raised when a literal operand in a fixed-point comparison or membership test is outside the range of the base type. (See test C45252A.)
- (6) Underflow is not gradual. (See tests C45524A..Z (26 tests).)

d. Rounding.

The method by which values are rounded in type conversions is not defined by the language. While the ACVC tests do not specifically attempt to determine the method of rounding, the test results indicate the following:

- (1) The method used for rounding to integer is round away from zero. (See tests C46012A..Z (26 tests).)
- (2) The method used for rounding to longest integer is round away from zero. (See tests C46012A..Z (26 tests).)
- (3) The method used for rounding to integer in static universal real expressions is round away from zero. (See test C4A014A.)

e. Array types.

An implementation is allowed to raise `NUMERIC_ERROR` or `CONSTRAINT_ERROR` for an array having a `'LENGTH` that exceeds `STANDARD.INTEGER'LAST` and/or `SYSTEM.MAX_INT`.

For this implementation:

- (1) Declaration of an array type or subtype declaration with more than `SYSTEM.MAX_INT` components raises `CONSTRAINT_ERROR`. (See test C36003A.)
- (2) `CONSTRAINT_ERROR` is raised when an array type with `INTEGER'LAST + 2` components with each component being a null array is declared. (See test C36202A.)
- (3) `CONSTRAINT_ERROR` is raised when an array type with `SYSTEM.MAX_INT + 2` components with each component being a null array is declared. (See test C36202B.)
- (4) A packed `BOOLEAN` array having a `'LENGTH` exceeding `INTEGER'LAST` raises `CONSTRAINT_ERROR` when the array type is declared. (See test C52103X.)

CONFIGURATION INFORMATION

- (5) A packed two-dimensional BOOLEAN array with more than INTEGER'LAST components raises CONSTRAINT_ERROR when the array type is declared. (See test C52104Y.)
- (6) A null array with one dimension of length greater than INTEGER'LAST may raise NUMERIC_ERROR or CONSTRAINT_ERROR either when declared or assigned. Alternatively, an implementation may accept the declaration. However, lengths must match in array slice assignments. This implementation raises CONSTRAINT_ERROR when the array type is declared. (See test E52103Y.)
- (7) In assigning one-dimensional array types, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)
- (8) In assigning two-dimensional array types, the expression is not evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

f. Discriminated types.

- (1) In assigning record types with discriminants, the expression is evaluated in its entirety before CONSTRAINT_ERROR is raised when checking whether the expression's subtype is compatible with the target's subtype. (See test C52013A.)

g. Aggregates.

- (1) In the evaluation of a multi-dimensional aggregate, all choices are evaluated before checking against the index type. (See tests C43207A and C43207B.)
- (2) In the evaluation of an aggregate containing subaggregates, not all choices are evaluated before being checked for identical bounds. (See test E43212B.)
- (3) CONSTRAINT_ERROR is raised after all choices are evaluated when a bound in a non-null range of a non-null aggregate does not belong to an index subtype. (See test E43211B.)

h. Pragmas.

- (1) The pragma INLINE is supported for functions and procedures. (See tests LA3004A..B (2 tests), EA3004C..D (2 tests), and CA3004E..F (2 tests).)

i. Generics.

- (1) Generic specifications and bodies can be compiled in separate compilations. (See tests CA1012A, CA2009C, CA2009F, BC3204C, and BC3205D.)
- (2) Generic subprogram declarations and bodies can be compiled in separate compilations. (See tests CA1012A and CA2009F.)
- (3) Generic library subprogram specifications and bodies can be compiled in separate compilations. (See test CA1012A.)
- (4) Generic non-library package bodies as subunits cannot be compiled in separate compilations. (See test CA2009C.)
- (5) Generic non-library subprogram bodies cannot be compiled in separate compilations from their stubs. (See test CA2009F.)
- (6) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)
- (7) Generic package declarations and bodies cannot be compiled in separate compilations. (See tests CA2009C, BC3204C, and BC3205D.)
- (8) Generic library package specifications and bodies cannot be compiled in separate compilations. (See tests BC3204C and BC3205D.)
- (9) Generic unit bodies and their subunits can be compiled in separate compilations. (See test CA3011A.)

j. Input and output.

- (1) The package SEQUENTIAL_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101C, EE2201D, and EE2201E.)
- (2) The package DIRECT_IO cannot be instantiated with unconstrained array types or record types with discriminants without defaults. (See tests AE2101H, EE2401D, and EE2401G.)
- (3) The Director, AJPO, has determined (AI-00332) that every call to OPEN and CREATE must raise USE ERROR or NAME ERROR if file input/output is not supported. This implementation exhibits this behavior for SEQUENTIAL_IO, DIRECT_IO, and TEXT_IO.

CHAPTER 3
TEST INFORMATION

3.1 TEST RESULTS

Version 1.10 of the ACVC comprises 3717 tests. When this compiler was tested, 44 tests had been withdrawn because of test errors. The AVF determined that 772 tests were inapplicable to this implementation. All inapplicable tests were processed during validation testing except for 285 executable tests that use floating-point precision exceeding that supported by the implementation and 242 executable tests that use file operations not supported by the implementation. Modifications to the code, processing, or grading for 6 tests were required to successfully demonstrate the test objective.

The AVF concludes that the testing results demonstrate acceptable conformity to the Ada Standard.

3.2 SUMMARY OF TEST RESULTS BY CLASS

RESULT	TEST CLASS						TOTAL
	A	B	C	D	E	L	
Passed	116	1129	1579	17	14	46	2901
Inapplicable	13	9	736	0	14	0	772
Withdrawn	1	2	35	0	6	0	44
TOTAL	130	1140	2350	17	34	46	3717

TFST INFORMATION

3.3 SUMMARY OF TEST RESULTS BY CHAPTER

RESULT	CHAPTER													TOTAL
	2	3	4	5	6	7	8	9	10	11	12	13	14	
Passed	192	532	496	244	172	99	159	331	135	36	250	179	76	2901
Inappl	20	117	184	4	0	0	7	1	2	0	2	190	245	772
Wdrn	1	1	0	0	0	0	0	2	0	0	1	35	4	44
TOTAL	213	650	680	248	172	99	166	334	137	36	253	404	325	3717

3.4 WITHDRAWN TESTS

The following 44 tests were withdrawn from ACVC Version 1.10 at the time of this validation:

E28005C	A39005G	B97102E	C97116A	BC3009B	CD2A62D
CD2A63A	CD2A63B	CD2A63C	CD2A63D	CD2A66A	CD2A66B
CD2A66C	CD2A66D	CD2A73A	CD2A73B	CD2A73C	CD2A73D
CD2A76A	CD2A76B	CD2A76C	CD2A76D	CD2A81G	CD2A83G
CD2A84M	CD2A84N	CD2B15C	CD2D11B	CD5007B	CD50110
ED7004B	ED7005C	ED7005D	ED7006C	ED7006D	CD7105A
CD7203B	CD7204B	CD7205C	CD7205D	CE2107I	CE3111C
CE3301A	CE3411B				

See Appendix D for the reason that each of these tests was withdrawn.

3.5 INAPPLICABLE TESTS

Some tests do not apply to all compilers because they make use of features that a compiler is not required by the Ada Standard to support. Others may depend on the result of another test that is either inapplicable or withdrawn. The applicability of a test to an implementation is considered each time a validation is attempted. A test that is inapplicable for one validation attempt is not necessarily inapplicable for a subsequent attempt. For this validation attempt, 772 tests were inapplicable for the reasons indicated:

- a. The following 285 tests are not applicable because they have floating-point type declarations requiring more digits than `SYSTEM.MAX_DIGITS`:

C24113F..Y (20 tests) C35705F..Y (20 tests) C35706F..Y (20 tests)
 C35707F..Y (20 tests) C35708F..Y (20 tests) C35802F..Z (21 tests)
 C45241F..Y (20 tests) C45321F..Y (20 tests) C45421F..Y (20 tests)

TEST INFORMATION

C45521F..Z (21 tests) C45524F..Z (21 tests) C45621F..Z (21 tests)
C45641F..Y (20 tests) C46012F..Z (21 tests)

- b. The following 50 tests are not applicable because this implementation does not support enumeration representation clauses:

C35502I..J (2 tests)	C35502M..N (2 tests)	C35507I..J (2 tests)
C35507M..N (2 tests)	A39005F	C55B16A
AD1009M	AD1009V..W (2 tests)	CD1C03G
AD1C04D	CD2A23A..E (5 tests)	CD2A24A..J (10 tests)
ED2A26A	CD3014A..B (2 tests)	AD3014C
CD3014D..E (2 tests)	AD3014F	CD3015A..B (2 tests)
AD3015C	CD3015D..E (2 tests)	AD3015F
CD3015G	AD3015H	CD3015I..J (2 tests)
AD3015K	CD3015L	CD3021A

- c. The following 31 tests are not applicable because this implementation does not support representation specifications for derived types:

C35508I..J (2 tests)	C35508M..N (2 tests)	C87B62A
CD1C04A..B (2 tests)	CD1C04E	CD2A21C..D (2 tests)
CD2A31C	CD2A41C..D (2 tests)	CD2A51C
CD2A64A	CD2A64C	CD2A65A
CD2A65C	CD2A74A	CD2A74C
CD2A75A	CD2A75C	CD2A81C..D (2 tests)
CD2A87A	CD2A91C..D (2 tests)	CD4051A..D (4 tests)

- d. C35702A and B86001T are not applicable because this implementation supports no predefined type SHORT_FLOAT.

- e. The following 100 tests are not applicable because they contain length clauses specifying the size of types or objects. The size specified is other than what the compiler would have chosen, and is rejected :

A39005B	CD1009A..C (3 tests)	CD1009H..I (2 tests)
CD10090	CD1C03A	CD2A21A..B (2 tests)
CD2A21E	CD2A22A..J (10 tests)	CD2A31A..B (2 tests)
CD2A31D	CD2A32A..J (10 tests)	CD2A41A..B (2 tests)
CD2A41E	CD2A42A..J (10 tests)	CD2A51A..B (2 tests)
CD2A51D..E (2 tests)	CD2A52A..D (4 tests)	CD2A52G..J (4 tests)
CD2A61A..L (12 tests)	CD2A62A..C (3 tests)	CD2A64B
CD2A64D	CD2A65B	CD2A65D
CD2A71A..D (4 tests)	CD2A72A..D (4 tests)	CD2A74B
CD2A74D	CD2A75B	CD2A75D
CD2A84B..I (8 tests)	CD2A84K..L (2 tests)	

- f. The following 21 tests are not applicable because this implementation does not support length clauses specifying 'SMALL for a fixed-point type:

TEST INFORMATION

A39005E	C87B62C	CD1009L
CD1C03F	CD1C04C	CD2A53A..E (5 tests)
CD2A54A..D (4 tests)	CD2A54G..J (4 tests)	ED2A56A
CD2D11A	CD2D13A	

- g. The following 16 tests are not applicable because this implementation does not support a predefined type `SHORT_INTEGER`:

C45231B	C45304B	C45502B	C45503B	C45504B
C45504E	C45611B	C45613B	C45614B	C45631B
C45632B	B52004E	C55B07B	B55B09D	B86001V
CD7101E				

- h. C45231D, B86001X, and CD7101G are not applicable because this implementation does not support any predefined integer type with a name other than `INTEGER`, `LONG_INTEGER`, or `SHORT_INTEGER`.
- i. C45531M..P (4 tests) and C45532M..P (4 tests) are not applicable because the value of `SYSTEM.MAX_MANTISSA` is less than 48.
- j. B86001Y is not applicable because this implementation supports no predefined fixed-point type other than `DURATION`.
- k. B86001Z is not applicable because this implementation supports no predefined floating-point type with a name other than `FLOAT`, `LONG_FLOAT`, or `SHORT_FLOAT`.
- l. C96005B is not applicable because there are no values of type `DURATION'BASE` that are outside the range of `DURATION`.
- m. CA2009C, CA2009F, BC3204C, and BC3204D are not applicable because this implementation does not permit compilation of generic specification and body when the body comes after in instantiation of the generic unit.
- n. CD4041A is not applicable because this implementation does not support the alignment clause with the record representation clause.
- o. CD5012J, CD5013S, and CD5014S are not applicable because this implementation does not support address clauses for a variable of a task type.

TEST INFORMATION

- p. The following 242 tests are inapplicable because sequential, text, and direct access files are not supported:

CE2102A..C (3 tests)	CE2102G..H (2 tests)	CE2102K
CE2102N..Y (12 tests)	CE2103C..D (2 tests)	CE2104A..D (4 tests)
CE2105A..B (2 tests)	CE2106A..B (2 tests)	CE2107A..H (8 tests)
CE2107L	CE2108A..B (2 tests)	CE2108C..H (6 tests)
CE2109A..C (3 tests)	CE2110A..D (4 tests)	CE2111A..I (9 tests)
CE2115A..B (2 tests)	CE2201A..C (3 tests)	EE2201D..E (2 tests)
CE2201F..N (9 tests)	CE2204A..D (4 tests)	CE2205A
CE2208B	CE2401A..C (3 tests)	EE2401D
CE2401E..F (2 tests)	EE2401G	CE2401H..L (5 tests)
CE2404A..B (2 tests)	CE2405B	CE2406A
CE2407A..B (2 tests)	CE2408A..B (2 tests)	CE2409A..B (2 tests)
CE2410A..B (2 tests)	CE2411A	CE3102A..B (2 tests)
EE3102C	CE3102F..H (3 tests)	CE3102J..K (2 tests)
CE3103A	CE3104A..C (3 tests)	CE3107B
CE3108A..B (2 tests)	CE3109A	CE3110A
CE3111A..B (2 tests)	CE3111D..E (2 tests)	CE3112A..D (4 tests)
CE3114A..B (2 tests)	CE3115A	EE3203A
CE3208A	EE3301B	CE3302A
CE3305A	CE3402A	EE3402B
CE3402C..D (2 tests)	CE3403A..C (3 tests)	CE3403E..F (2 tests)
CE3404B..D (3 tests)	CE3405A	EE3405B
CE3405C..D (2 tests)	CE3406A..D (4 tests)	CE3407A..C (3 tests)
CE3408A..C (3 tests)	CE3409A	CE3409C..E (3 tests)
EE3409F	CE3410A	CE3410C..E (3 tests)
EE3410F	CE3411A	CE3411C
CE3412A	EE3412C	CE3413A
CE3413C	CE3602A..D (4 tests)	CE3603A
CE3604A..B (2 tests)	CE3605A..E (5 tests)	CE3606A..B (2 tests)
CE3704A..F (6 tests)	CE3704M..O (3 tests)	CE3706D
CE3706F..G (2 tests)	CE3804A..P (16 tests)	CE3805A..B (2 tests)
CE3806A..B (2 tests)	CE3806D..E (2 tests)	CE3806G..H (2 tests)
CE3905A..C (3 tests)	CE3905L	CE3906A..C (3 tests)
CE3906E..F (2 tests)		

- q. CE2103A, CE2103B, and CE3107A were ruled not applicable by the AVO because this implementation raises USE_ERROR vice the expected NAME_ERROR.

TEST INFORMATION

3.6 TEST, PROCESSING, AND EVALUATION MODIFICATIONS

It is expected that some tests will require modifications of code, processing, or evaluation in order to compensate for legitimate implementation behavior. Modifications are made by the AVF in cases where legitimate implementation behavior prevents the successful completion of an (otherwise) applicable test. Examples of such modifications include: adding a length clause to alter the default size of a collection; splitting a Class B test into subtests so that all errors are detected; and confirming that messages produced by an executable test demonstrate conforming behavior that wasn't anticipated by the test (such as raising one exception instead of another).

Modifications were required for 6 tests.

The following tests were split because syntax errors at one point resulted in the compiler not detecting other errors in the test:

B33301B B55A01A BC1109A BC1109C BC1109D

Test AD7006A was modified because the Type SYSTEM.MEMORY_SIZE needed to be explicitly converted to type INTEGER. The modification was made so that line 23 was changed to:

```
I := INTEGER (SYSTEM.MEMORY_SIZE - MYMSIZE + 1);
```

3.7 ADDITIONAL TESTING INFORMATION

3.7.1 Prevalidation

Prior to validation, a set of test results for ACVC Version 1.10 produced by the DDC-Based Ada/CAPS Compiler, Version 4.1 was submitted to the AVF by the applicant for review. Analysis of these results demonstrated that the compiler successfully passed all applicable tests, and the compiler exhibited the expected behavior on all inapplicable tests.

3.7.2 Test Method

Testing of the DDC-Based Ada/CAPS Compiler, Version 4.1 using ACVC Version 1.10 was conducted on-site by a validation team from the AVF. The configuration in which the testing was performed is described by the following designations of hardware and software components:

Host computer:	VAXstation 3100 Model 30
Host operating system:	VMS, 5.2
Target computer:	CAPS/AAMP2
Target operating system:	bare machine
Compiler:	DDC-Based Ada/CAPS Compiler, Version 4.1

TEST INFORMATION

The host and target computers were linked via Ethernet.

A magnetic tape containing all tests except for withdrawn tests and tests requiring unsupported floating-point precisions was taken on-site by the validation team for processing. Tests that make use of implementation-specific values were customized before being written to the magnetic tape. Tests requiring modifications during the prevalidation testing were included in their modified form on the magnetic tape.

The contents of the magnetic tape were loaded first onto a temporary directory on a VAX 8700, then copied via Ethernet to the VAXstation network fileserver.

After the test files were loaded to disk, the full set of tests was compiled and linked on two VAXstation 3100 Model 30s; then all executable images were transferred to the CAPS/AAMP2 via Ethernet and run. Results were first transferred back to the host computer, then copied to the 8700 to be printed.

The compiler was tested using command scripts provided by Rockwell International Corporation and reviewed by the validation team. The compiler was tested using all the following option settings. See Appendix E for a complete listing of the compiler options for this implementation. The following list of compiler options includes those options which were invoked by default:

/LIST	Write source listing to the list file
/NODEBUG	Suppress generation of Debugger Symbol Table files.
/OPTIMIZE	Various target-independent optimizations and a peephole optimizer are enabled.

Tests were compiled and linked using two host computers, and executed using one target computer. Test output, compilation listings, and job logs were captured on magnetic tape and archived at the AVF. The listings examined on-site by the validation team were also archived.

TEST INFORMATION

3.7.3 Test Site

Testing was conducted at Cedar Rapids IA and was completed on 01 November 1989.

APPENDIX A

DECLARATION OF CONFORMANCE

Rockwell International Corporation has submitted the following Declaration of Conformance concerning the DDC-Based Ada/CAPS Compiler, Version 4.1.

Declaration of Conformance

Customer: Rockwell International Corporation
Ada Validation Facility: ASD/SCEL, Wright-Patterson AFB, OH
ACVC Version: 1.10

Ada Implementation

Ada Compiler Name: DDC-Based Ada/CAPS Compiler System
Version: 4.1
Host Computer System: VAXstation 3100 Model 30
Host OS & Version: VMS 5.2
Target Computer System: CAPS/AAMP2
Target OS & Version: Bare Machine

Customer's Declaration

I, the undersigned, representing Rockwell International Corporation declare that Rockwell International Corporation has no knowledge of deliberate deviations from the Ada Language Standard ANSI/MIL-STD-1815A in the implementation(s) listed in this declaration.

C. E. Kress
C. E. Kress,
Manager of Processor Technology Department
Rockwell International Corporation

10/30/89
Date

APPENDIX B

APPENDIX F OF THE Ada STANDARD

The only allowed implementation dependencies correspond to implementation-dependent pragmas, to certain machine-dependent conventions as mentioned in Chapter 13 of the Ada Standard, and to certain allowed restrictions on representation clauses. The implementation-dependent characteristics of the DDC-Based Ada/CAPS Compiler, Version 4.1, as described in this Appendix, are provided by Rockwell International Corporation. Unless specifically noted otherwise, references in this Appendix are to compiler documentation and not to this report. Implementation-specific portions of the package STANDARD, which are not a part of Appendix F, are:

package STANDARD is

...

type INTEGER is range -32767 .. 32768;

type LONG_INTEGER is range -2_147_483_648 .. 2_147_483_647;

type FLOAT is digits 6 range -16#0.7FFF_FF8#E+32 .. 16#0.7FFF_FF8#E+32;

type LONG_FLOAT is digits 9 range -16#0.7FFF_FFFF_FF8#E+32 ..
16#0.7FFF_FFFF_FF8#E+32;

type DURATION is delta 0.0001 range -131072.0000 .. 131071.999938965;

...

end STANDARD;

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

This appendix describes the implementation-dependent characteristics of the DDC-Based Ada/CAPS Compiler.

F.1 Implementation-Dependent Pragas.

F.1.1 Pragma EXPORT.

Takes an identifier denoting a subprogram or an object, and optionally takes a string literal (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram or object declared by an earlier declarative item in the same declarative part or package specification. The pragma must occur in the same compilation unit as the subprogram body to export a subprogram, and in the same compilation unit as the declaration to export an object. The subprogram to be exported may not be nested within anything but a library_unit package specification or body. The pragma is not allowed for an access or a task object. The object exported must be a static object. Generally, objects declared in a package specification or body are static; objects declared local to a subprogram are not.

This pragma allows the export of a procedure, function, or object to a non-Ada environment.

```
pragma EXPORT(internal_name [, external_name]);  
  
internal_name ::= identifier  
  
external_name ::= string_literal
```

If `external_name` is not specified, the `internal_name` is used as the `external_name`. If a `string_literal` is given, it is used. `External_name` must be an identifier that is acceptable to the CAPS linker, though it does not have to be a valid Ada identifier.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Exporting Subprograms:

In this case the pragma specifies that the body of the specified subprogram associated with an Ada subprogram specification may be called from another CAPS language (e.g., Jovial, or assembly).

Subprograms must be uniquely identified by their internal names. An exported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). An EXPORT pragma is allowed for a subprogram which is a compilation unit only after the subprogram body in the compilation unit. It is allowed for a subprogram in a package body after the body of the subprogram. Pragma EXPORT is not allowed in a package specification.

Example:

```
procedure AUTO_PILOT (MODE: in INTEGER) is
    ...
end AUTO_PILOT;
pragma EXPORT (AUTO_PILOT);
```

Exporting Objects:

In this case the pragma specifies that an Ada object is to be accessible by an external routine in an another CAPS language.

Objects must be uniquely identified by their internal names. An exported object must be a variable declared in the outermost declarative part of a library package (specification or body).

The object must be allocated to static storage. To guarantee this, the subtype indication for the object must denote one of the following:

- o A scalar type or subtype.
- o An array subtype with static index constraints whose component size is static.
- o A simple record type or subtype.

Example:

```
SYSTEM_STATUS: INTEGER;
pragma_EXPORT (SYSTEM_STATUS, "SYSSSTS");  -- SYSSSTS is a Jovial
                                           -- identifier.
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.1.2 Pragma IMPORT.

Takes an internal name denoting a subprogram, and optionally takes an external name (the name of a CAPS object module entry/external name) as arguments. This pragma is only allowed at the place of a declarative item and must apply to a subprogram declared by an earlier declaration item in the same declarative part or package specification.

This pragma allows the import of a procedure or function from a non-Ada environment.

```
pragma IMPORT (internal_name [, external_name]);  
  
internal_name ::= identifier | string_literal  
  
external_name ::= identifier | string_literal
```

Internal_name may only be a string_literal when designating an operator_function for import. If external_name is not specified, the internal_name is used as the external_name. If an identifier or string_literal is given, it is used. External_name must name an identifier that is acceptable to the CAPS target linker though it does not have to be a valid Ada identifier.

Importing Subprograms:

In this case the pragma specifies that the body of the specified subprogram associated with an Ada subprogram specification is to be provided by another CAPS language. The pragma INTERFACE must also be given for the internal_name earlier for the same declarative part or package specification. The use of the pragma INTERFACE implies that a corresponding body is not given.

Subprograms must be uniquely identified by their internal names. An imported subprogram must be a library unit or be declared in the outermost declarative part of a library package (specification or body). An import pragma is allowed only if either the body does not have a corresponding specification, or the specification and body occur in the same declarative part.

If a subprogram has been declared as a compilation unit, the pragma is only allowed after the subprogram declaration and before any subsequent compilation unit. This pragma may not be used for a subprogram that is declared by a generic instantiation of a predefined subprogram.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Example:

```
function SIN (X: in FLOAT) return FLOAT;
pragma INTERFACE (ASSEMBLY, SIN);
pragma IMPORT (SIN, "SIN$$");
```

F.1.3 Pragma STACK_SIZE.

This pragma has two arguments, a task type name and an integer expression. This pragma is allowed anywhere that a task storage specification is allowed. The effect of this pragma is to use the value of the expression as the number of storage units (words) to be allocated to the process stack of tasks of the associated task type.

Example:

```
task type DISPLAY_UNIT is
    entry UPPER_DISPLAY;
    entry BOTTOM_LINE;
end DISPLAY_UNIT;

for DISPLAY_UNIT'SORAGE_SIZE use 20_000;      -- Data Stack.
pragma STACK_SIZE (DISPLAY_UNIT, 1000);      -- Process Stack.
```

F.1.4 Pragma UNIVERSAL_DATA

This pragma allows the static data of a library package specification or body to be located anywhere in memory instead of the first 64K words. Use of this pragma causes less efficient code (in both space and speed) to be generated by the compiler. Specifications and bodies of the same unit are treated independently and the pragma applies only to the compilation unit it appears in.

The pragma must appear in the declarative part of a package specification or body, and must appear before any other object declaration.

The functionality of this pragma is also available as the compiler command option /PRAGMA = UNIVERSAL_DATA.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Example:

```
package AUTO_PILOT is
  pragma UNIVERSAL_DATA;

  -- object declarations

  ...

end AUTO_PILOT;
```

F.2 Implementation-Dependent Attributes.

No implementation-dependent attributes are supported.

F.3 Specification Of The Package SYSTEM.

package SYSTEM is

```
  type ADDRESS      is range 0..16#FF FFFF#    -- 24 bit address
  subtype PRIORITY  is INTEGER range 1 .. 254;
  type NAME          is (AAMP1, AAMP2);
  SYSTEM_NAME:      constant NAME      := AAMPx;
  STORAGE_UNIT:     constant           := 16;
  MEMORY_SIZE:      constant           := 16_384 * 1024;
  MIN_INT:          constant           := -2147_483_647-1;
  MAX_INT:          constant           := 2147_483_647;
  MAX_DIGITS:       constant           := 9;
  MAX_MANTISSA:     constant           := 31;
  FINE_DELTA:       constant           := 2#1.0#E-31;
  TICK:             constant           := 0.000_1;
```

```
  type INTERFACE_LANGUAGE is (ASSEMBLY);
```

```
end SYSTEM;
```

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.4 Representation Clause Restrictions.

F.4.1 Representation Clauses.

In general, no representation clauses may given for a derived type. The representation clauses that are allowed for non-derived types are described in the following sections.

F.4.2 Length Clauses.

The compiler accepts a length clause that specifies the number of storage units reserved for a collection or for a task data stack size, via the 'STORAGE SIZE clause. (See pragma STACK SIZE for a complementary capability.) The 'SIZE clause has no effect for tasks.

A 'SIZE clause for a type is rejected if the given size is not equal to the size the compiler would have chosen for the type. It serves as a compile-time assertion, to check the user's assumptions as to how the compiler allocates storage.

F.4.3 Enumeration Representation Clauses.

Enumeration representation clauses are not supported.

F.5 Implementation-Generated Names.

Implementation-generated names for implementation-dependent components are not supported.

F.6 Address Clause Expressions.

All address values are interpreted as the 24-bit address of a 16 bit word of memory, even for code addresses which are normally thought of as byte addresses. All subprogram and task entry addresses are word aligned by the compiler.

F.7 Unchecked Conversion Restrictions.

Unchecked conversion is only allowed between objects of the same size.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.8 I/O Package Implementation-Dependent Characteristics.

The target environment does not support a file system; therefore I/O procedure or function calls involving files (except STANDARD_OUTPUT, etc_. as noted below) will raise an exception.

F.8.1 Package SEQUENTIAL_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN which raise USE_ERROR, and IS_OPEN which always returns FALSE.

F.8.2 Package DIRECT_IO.

All procedures and functions raise STATUS_ERROR, except for CREATE and OPEN which raise USE_ERROR, and IS_OPEN which always returns FALSE.

F.8.3 Package TEXT_IO.

No disk file system is supported. Therefore procedures CREATE and OPEN always raise USE_ERROR.

The output routines with no file parameter, which operate on the current default output file, are implemented and produce their output via package LOW_IO. Since no external files can be opened, the output routines with a file parameter raise STATUS_ERROR unless the actual parameter is one of the functions STANDARD_OUTPUT or CURRENT_OUTPUT.

Similarly, the input routines with a file parameter raise STATUS_ERROR unless the parameter is STANDARD_INPUT or CURRENT_INPUT.

Function IS_OPEN returns TRUE if the parameter is one of STANDARD_INPUT, STANDARD_OUTPUT, CURRENT_INPUT, or CURRENT_OUTPUT; otherwise FALSE is returned.

F.8.4 Package LOW_LEVEL_IO.

Package LOW_LEVEL_IO is not provided.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

F.8.5 Package LOW_IO.

Package LOW_IO is used by TEXT_IO for character-level I/O operations. The implementation provided with the compiler system sends output messages to the Symbolic Debugger to be printed on the screen and optionally sent to a .LOG file. The Debugger does not implement input.

The body of LOW_IO can be replaced by the user, for example to communicate with a terminal via an RS-232 port.

F.9 Other Implementation-Dependent Features.

F.9.1 Predefined Types.

This section describes the implementation-dependent predefined types declared in the predefined package STANDARD, and the relevant attributes of these types.

F.9.1.1 Integer Types.

Two predefined integer types are implemented, INTEGER, and LONG_INTEGER. They have the following attributes:

INTEGER'FIRST	=	-32768
INTEGER'LAST	=	32767
INTEGER'SIZE	=	16
LONG_INTEGER'FIRST	=	-2_147_483_648
LONG_INTEGER'LAST	=	2_147_483_647
LONG_INTEGER'SIZE	=	32

F.9.1.2 Floating Point Types.

Two predefined floating point types are implemented, FLOAT and LONG_FLOAT. They have the following attributes:

FLOAT'DIGITS	=	6
FLOAT'EMAX	=	84
FLOAT'EPSILON	=	16#0.1000_000#E-04
	~	= 9.53674E-07
FLOAT'FIRST	=	-16#0.7FFF_FF8#E+32
	~	= -1.70141E+38
FLOAT'LARGE	=	16#0.FFFF_FF80#E+21
	~	= 1.93428E+25
FLOAT'LAST	=	16#0.7FFF_FF8#E+32
	~	= 1.70141E+38

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

FLOAT'MACHINE_EMAX	=	127
FLOAT'MACHINE_EMIN	=	-127
FLOAT'MACHINE_MANTISSA	=	24
FLOAT'MACHINE_OVERFLOWS	=	TRUE
FLOAT'MACHINE_RADIX	=	2
FLOAT'MACHINE_ROUNDS	=	TRUE
FLOAT'MANTISSA	=	21
FLOAT'SAFE_EMAX	=	127
FLOAT'SAFE_LARGE	=	16#0.7FFF_FC#E+32
	~	1.70141E+38
FLOAT'SAFE_SMALL	=	16#0.1000_000#E-31
	~	2.93874E-39
FLOAT'SIZE	=	32
FLOAT'SMALL	=	16#0.8000_000#E-21
	~	2.58494E-26
LONG_FLOAT'DIGITS	=	9
LONG_FLOAT'EMAX	=	124
LONG_FLOAT'EPSILON	=	16#0.4000_0000_000#E-7
	~	9.31322575E-10
LONG_FLOAT'FIRST	=	-16#0.7FFF_FFFF_FF8#E+32
	~	-1.70141183E+38
LONG_FLOAT'LARGE	=	16#0.FFFF_FFFE_000#E+31
	~	2.12676479E+37
LONG_FLOAT'LAST	=	16#0.7FFF_FFFF_FF8#E+32
	~	1.70141183E+38
LONG_FLOAT'MACHINE_EMAX	=	127
LONG_FLOAT'MACHINE_EMIN	=	-127
LONG_FLOAT'MACHINE_MANTISSA	=	40
LONG_FLOAT'MACHINE_OVERFLOWS	=	TRUE
LONG_FLOAT'MACHINE_RADIX	=	2
LONG_FLOAT'MACHINE_ROUNDS	=	TRUE
LONG_FLOAT'MANTISSA	=	31
LONG_FLOAT'SAFE_EMAX	=	127
LONG_FLOAT'SAFE_LARGE	=	16#0.7FFF_FFFF#E+32
	~	1.70141183E+38
LONG_FLOAT'SAFE_SMALL	=	16#0.1000_0000_000#E-31
	~	2.93873588E-39
LONG_FLOAT'SIZE	=	48
LONG_FLOAT'SMALL	=	16#0.8000_0000_000#E-31
	~	2.35098870E-38

F.9.1.3 Fixed Point Types.

To implement fixed point numbers, Ada/CAPS uses two sets of anonymous, predefined, fixed point types, here named `FIXED` and `LONG_FIXED`. These names are not defined in package `STANDARD`, but are only used here for reference.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

These types are of the following form:

type `FIXED_TYPE` is delta `SMALL` range $-M \cdot \text{SMALL} \dots (M-1) \cdot \text{SMALL}$;

where $\text{SMALL} = 2^n$ for $-128 \leq n \leq 127$,

and $M = 2^{15}$ for `FIXED`, or $M = 2^{31}$ for `LONG_FIXED`.

For each of `FIXED` and `LONG_FIXED` there exists a virtual predefined type for each possible value of `SMALL` (cf. RM 3.5.9). `SMALL` may be any power of 2 which is representable by a `LONG_FLOAT` value. `FIXED` types are represented by 16 bits, while 32 bits are used for `LONG_FIXED` types.

A user-defined fixed point type is represented as that predefined `FIXED` or `LONG_FIXED` type which has the largest value of `SMALL` not greater than the user-specified `DELTA`, and which has the smallest range that includes the user-specified range.

As the value of `SMALL` increases, the range increases. In other words, the greater the allowable error (the value of `SMALL`), the larger the allowable range.

Example 1:

For a 16-bit `FIXED` type, to get the smallest amount of error possible requires $\text{SMALL} = 2^{(-128)}$, but the range is constrained to:

$$\begin{aligned} &-(2^{15}) * 2^{(-128)} \dots (2^{15} - 1) * 2^{(-128)}, \text{ which is} \\ &-(2^{(-113)}) \dots 2^{(-113)} - 2^{(-128)}. \end{aligned}$$

Example 2:

For a `FIXED` type, to get the largest range possible requires $\text{SMALL} = 2^{127}$, i.e., the error may be as large as 2^{127} . The range is then:

$$\begin{aligned} &-(2^{15}) * (2^{127}) \dots (2^{15} - 1) * (2^{127}), \text{ which is} \\ &-(2^{142}) \dots (2^{142}) - (2^{127}). \end{aligned}$$

Example 3:

Two particularly useful fixed-point types range from -1.0 to 1.0 - delta. The compiler is able to take advantage of special AAMP instructions to generate more efficient code for multiplication and division than for fixed-point types in general.

type `FRACT` is delta $2.0^{(-15)}$ range -1.0 .. 1.0 - $2.0^{(-15)}$;

type `LONG_FRACT` is delta $2.0^{(-31)}$ range -1.0 .. 1.0 - $2.0^{(-31)}$;

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

For any FIXED or LONGFIXED type T:

```
T'MACHINE_OVERFLOWS = TRUE
T'MACHINE_ROUNDS    = FALSE
```

F.9.1.4 The Type DURATION.

The predefined fixed point type DURATION has the following attributes:

```
DURATION'AFT          = 4
DURATION'DELTA        = 0.0001
DURATION'FIRST        = -131072.0000
DURATION'FORE         = 7
DURATION'LARGE        = 131071.999938965
                     = 2#1.0#E+17 - 2#1.0E-14
DURATION'LAST         = DURATION'LARGE
DURATION'MANTISSA     = 31
DURATION'SAFELARGE    = DURATION'LARGE
DURATION'SAFESMALL    = DURATION'SMALL
DURATION'SIZE         = 32
DURATION'SMALL        = 6.103515625E-5
                     = 2#1.0#E-14
```

F.9.2 Uninitialized Variables.

There is no check on the use of uninitialized variables. The effect of a program that uses the value of such a variable is undefined.

F.9.3 Package MACHINE_CODE.

Machine code insertions (cf. RM 13.8) are supported by the Ada/CAPS compiler via the use of the predefined package MACHINE_CODE.

package MACHINE_CODE is

```
    type CODE is record
        INSTR: STRING (1 .. 80);
    end record;
```

end MACHINE_CODE;

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

Machine code insertions may be used only in a procedure body. No local declarations are allowed, except for USE clauses. The body must contain nothing but code statements, as in the following example:

```
with MACHINE_CODE; -- Must apply to the compilation unit
                    -- containing DOUBLE.

procedure DOUBLE (VALUE: in INTEGER; DOUBLED_VALUE: out INTEGER) is
  use MACHINE_CODE;
begin
  CODE' (INSTR => "    REFSL    1    ;"); -- Get VALUE.
  CODE' (INSTR => "    DUP      ;"); -- Make a copy of VALUE.
  CODE' (INSTR => "    ADD      ;"); -- Add copies together.
  CODE' (INSTR => "    ASNSL    0    ;"); -- Store result in
                                         -- DOUBLED_VALUE.
end DOUBLE;
```

The string literal assigned to INSTR may be any CAPS assembly language instruction or macro. The string is written directly to the assembly output file.

The file AAMPx.MAC (where x is 1 or 2), located in the TOOLS subdirectory of the compiler system, defines the macros and instructions which are available for use. The macros may change with different compiler system releases and should be used cautiously, as there is no guarantee that they will perform the same across all releases. The CAPS Macro Assembler User's Guide contains information on how to call macros and write assembly instructions.

Several application notes are available which explain in greater detail how to use machine-code insertions, including clever ways to implement built-in functions.

F.9.4 Compiler Limitations.

The following limitations apply to Ada programs in the DDC-Based Ada/CAPS Compiler System:

- o A compilation unit may not contain more than 64K bytes (32K words) of code.
- o A compilation unit may not contain more than 32K words of data.
- o A compilation unit may not contain more than 32K words of constants.
- o It follows that any single object may be no larger than 32K words.

IMPLEMENTATION-DEPENDENT CHARACTERISTICS

- o No more than 500 subprograms may be defined in a single compilation unit, including any implicitly allocated by the compiler.
- o The maximum nesting level for blocks is 100.

APPENDIX C TEST PARAMETERS

Certain tests in the ACVC make use of implementation-dependent values, such as the maximum length of an input line and invalid file names. A test that makes use of such values is identified by the extension .TST in its file name. Actual values to be substituted are represented by names that begin with a dollar sign. A value must be substituted for each of these names before the test is run. The values used for this validation are given below.

<u>Name and Meaning</u>	<u>Value</u>
\$ACC SIZE An integer literal whose value is the number of bits sufficient to hold any value of an access type.	32
\$BIG ID1 An identifier the size of the maximum input line length which is identical to \$BIG ID2 except for the last character.	(1..125 => 'A', 126 => '1')
\$BIG ID2 An identifier the size of the maximum input line length which is identical to \$BIG ID1 except for the last character.	(1..125 => 'A', 126 => '2')
\$BIG ID3 An identifier the size of the maximum input line length which is identical to \$BIG ID4 except for a character near the middle.	(1..62 => 'A', 63 => '3', 64..126 => 'A')

TEST PARAMETERS

Name and Meaning	Value
\$BIG_ID4 An identifier the size of the maximum input line length which is identical to \$BIG_ID3 except for a character near the middle.	(1..62 => 'A', 63 => '4', 64..126 => 'A')
\$BIG_INT_LIT An integer literal of value 298 with enough leading zeroes so that it is the size of the maximum line length.	(1..123 => '0', 124..126 => "298")
\$BIG_REAL_LIT A universal real literal of value 690.0 with enough leading zeroes to be the size of the maximum line length.	(1..120 => '0', 121..126 => "69.0E1")
\$BIG_STRING1 A string literal which when catenated with \$BIG_STRING2 yields the image of \$BIG_ID1.	(1 => '"', 2..64 => 'A', 65 => '"')
\$BIG_STRING2 A string literal which when catenated to the end of \$BIG_STRING1 yields the image of \$BIG_ID1.	(1 => '"', 2..63 => 'A', 64 => '1', 65 => '"')
\$BLANKS A sequence of blanks twenty characters less than the size of the maximum line length.	(1..106 => ' ')
\$COUNT_LAST A universal integer literal whose value is TEXT_IO.COUNT'LAST.	2147483647
\$DEFAULT_MEM_SIZE An integer literal whose value is SYSTEM.MEMORY_SIZE.	16777216
\$DEFAULT_STOR_UNIT An integer literal whose value is SYSTEM.STORAGE_UNIT.	16

TEST PARAMETERS

Name and Meaning	Value
\$DEFAULT_SYS NAME The value of the constant SYSTEM.SYSTEM_NAME.	AAMP2
\$DELTA_DOC A real literal whose value is SYSTEM.FINE_DELTA.	2#1.0#E-31
\$FIELD_LAST A universal integer literal whose value is TEXT_IO.FIELD'LAST.	35
\$FIXED NAME The name of a predefined fixed-point type other than DURATION.	NO_SUCH_TYPE
\$FLOAT NAME The name of a predefined floating-point type other than FLOAT, SHORT_FLOAT, or LONG_FLOAT.	NO_SUCH_TYPE
\$GREATER_THAN_DURATION A universal real literal that lies between DURATION'BASE'LAST and DURATION'LAST or any value in the range of DURATION.	76536.0
\$GREATER_THAN_DURATION_BASE_LAST A universal real literal that is greater than DURATION'BASE'LAST.	10000000.0
\$HIGH_PRIORITY An integer literal whose value is the upper bound of the range for the subtype SYSTEM.PRIORITY.	254
\$ILLEGAL_EXTERNAL_FILE_NAME1 An external file name which contains invalid characters.	NO_SUCH_NAME1
\$ILLEGAL_EXTERNAL_FILE_NAME2 An external file name which is too long.	NO_SUCH_NAME2
\$INTEGER_FIRST A universal integer literal whose value is INTEGER'FIRST.	-32768

TEST PARAMETERS

Name and Meaning	Value
\$INTEGER LAST A universal integer literal whose value is INTEGER'LAST.	32767
\$INTEGER LAST PLUS 1 A universal integer literal whose value is INTEGER'LAST + 1.	32768
\$LESS THAN DURATION A universal real literal that lies between DURATION'BASE'FIRST and DURATION'FIRST or any value in the range of DURATION.	-76536.0
\$LESS THAN DURATION BASE FIRST A universal real literal that is less than DURATION'BASE'FIRST.	-10000000.0
\$LOW PRIORITY An integer literal whose value is the lower bound of the range for the subtype SYSTEM.PRIORITY.	1
\$MANTISSA DOC An integer literal whose value is SYSTEM.MAX_MANTISSA.	31
\$MAX DIGITS Maximum digits supported for floating-point types.	9
\$MAX IN LEN Maximum input line length permitted by the implementation.	126
\$MAX INT A universal integer literal whose value is SYSTEM.MAX_INT.	2147483647
\$MAX INT PLUS 1 A universal integer literal whose value is SYSTEM.MAX_INT+1.	2147483648
\$MAX LEN INT BASED LITERAL A universal integer based literal whose value is 2#11# with enough leading zeroes in the mantissa to be \$MAX_IN_LEN long.	(1..2 => "2:", 3..123 => '0', 124..126 => "11:")

TEST PARAMETERS

Name and Meaning	Value
\$MAX_LEN_REAL_BASED_LITERAL A universal real based literal whose value is 16:F.E: with enough leading zeroes in the mantissa to be \$MAX_IN_LEN long.	(1..3 => "16:", 4..122 => '0', 123..126 => "F.E:")
\$MAX_STRING_LITERAL A string literal of size \$MAX_IN_LEN, including the quote characters.	(1 => '"', 2..125 => 'A', 126 => '"')
\$MIN_INT A universal integer literal whose value is SYSTEM.MIN_INT.	-247147483648
\$MIN_TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has no entries, no declarations, and "NULL;" as the only statement in its body.	32
\$NAME A name of a predefined numeric type other than FLOAT, INTEGER, SHORT_FLOAT, SHORT_INTEGER, LONG_FLOAT, or LONG_INTEGER.	NO_SUCH_TYPE
\$NAME_LIST A list of enumeration literals in the type SYSTEM.NAME, separated by commas.	AAMP1,AAMP2
\$NEG_BASED_INT A based integer literal whose highest order nonzero bit falls in the sign bit position of the representation for SYSTEM.MAX_INT.	16#FFFFFFFE#
\$NEW_MEM_SIZE An integer literal whose value is a permitted argument for pragma MEMORY_SIZE, other than \$DEFAULT_MEM_SIZE. If there is no other value, then use \$DEFAULT_MEM_SIZE.	16777216

TEST PARAMETERS

Name and Meaning	Value
\$NEW STOR UNIT An integer literal whose value is a permitted argument for pragma STORAGE UNIT, other than \$DEFAULT_STOR_UNIT. If there is no other permitted value, then use value of SYSTEM.STORAGE_UNIT.	16
\$NEW SYS NAME A value of the type SYSTEM.NAME, other than \$DEFAULT_SYS_NAME. If there is only one value of that type, then use that value.	AAMP2
\$TASK_SIZE An integer literal whose value is the number of bits required to hold a task object which has a single entry with one 'IN OUT' parameter.	32
\$TICK A real literal whose value is SYSTEM.TICK.	0.0001

APPENDIX D
WITHDRAWN TESTS

Some tests are withdrawn from the ACVC because they do not conform to the Ada Standard. The following 44 tests had been withdrawn at the time of validation testing for the reasons indicated. A reference of the form AI-ddddd is to an Ada Commentary.

- a. E28005C: This test expects that the string "-- TOP OF PAGE. --63" of line 204 will appear at the top of the listing page due to a pragma PAGE in line 203; but line 203 contains text that follows the pragma, and it is this text that must appear at the top of the page.
- b. A39005G: This test unreasonably expects a component clause to pack an array component into a minimum size (line 30).
- c. B97102E: This test contains an unintended illegality: a select statement contains a null statement at the place of a selective wait alternative (line 31).
- d. C97116A: This test contains race conditions, and it assumes that guards are evaluated indivisibly. A conforming implementation may use interleaved execution in such a way that the evaluation of the guards at lines 50 & 54 and the execution of task CHANGING OF THE_GUARD results in a call to REPORT.FAILED at one of lines 52 or 56.
- e. BC3009B: This test wrongly expects that circular instantiations will be detected in several compilation units even though none of the units is illegal with respect to the units it depends on; by AI-00256, the illegality need not be detected until execution is attempted (line 95).
- f. CD2A62D: This test wrongly requires that an array object's size be no greater than 10 although its subtype's size was specified to be 40 (line 137).

WITHDRAWN TESTS

- g. CD2A63A..D, CD2A66A..D, CD2A73A..D, and CD2A76A..D (16 tests): These tests wrongly attempt to check the size of objects of a derived type (for which a 'SIZE length clause is given) by passing them to a derived subprogram (which implicitly converts them to the parent type (Ada standard 3.4:14)). Additionally, they use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- h. CD2A81G, CD2A83G, CD2A84M..N, and CD50110 (5 tests): These tests assume that dependent tasks will terminate while the main program executes a loop that simply tests for task termination; this is not the case, and the main program may loop indefinitely (lines 74, 85, 86, 96, and 58, respectively).
- i. CD2B15C and CD7205C: These tests expect that a 'STORAGE_SIZE length clause provides precise control over the number of designated objects in a collection; the Ada standard 13.2:15 allows that such control must not be expected.
- j. CD2D11B: This test gives a SMALL representation clause for a derived fixed-point type (at line 30) that defines a set of model numbers that are not necessarily represented in the parent type; by Commentary AI-00099, all model numbers of a derived fixed-point type must be representable values of the parent type.
- k. CD5007B: This test wrongly expects an implicitly declared subprogram to be at the address that is specified for an unrelated subprogram (line 303).
- l. ED7004B, ED7005C..D, and ED7006C..D (5 tests): These tests check various aspects of the use of the three SYSTEM pragmas; the AV0 withdraws these tests as being inappropriate for validation.
- m. CD7105A: This test requires that successive calls to CALENDAR.CLOCK change by at least SYSTEM.TICK; however, by Commentary AI-00201, it is only the expected frequency of change that must be at least SYSTEM.TICK--particular instances of change may be less (line 29).
- n. CD7203B and CD7204B: These tests use the 'SIZE length clause and attribute, whose interpretation is considered problematic by the WG9 ARG.
- o. CD7205D: This test checks an invalid test objective: it treats the specification of storage to be reserved for a task's activation as though it were like the specification of storage for a collection.

WITHDRAWN TESTS

- p. CE2107I: This test requires that objects of two similar scalar types be distinguished when read from a file--DATA_ERROR is expected to be raised by an attempt to read one object as of the other type. However, it is not clear exactly how the Ada standard 14.2.4:4 is to be interpreted; thus, this test objective is not considered valid (line 90).
- q. CE3111C: This test requires certain behavior, when two files are associated with the same external file, that is not required by the Ada standard.
- r. CE3301A: This test contains several calls to END_OF_LINE and END_OF_PAGE that have no parameter: these calls were intended to specify a file, not to refer to STANDARD_INPUT (lines 103, 107, 118, 132, and 136).
- s. CE3411B: This test requires that a text file's column number be set to COUNT'LAST in order to check that LAYOUT_ERROR is raised by a subsequent PUT operation. But the former operation will generally raise an exception due to a lack of available disk space, and the test would thus encumber validation testing.

APPENDIX E

COMPILER OPTIONS AS SUPPLIED BY
ROCKWELL INTERNATIONAL CORPORATION

Compiler: DDC-Based Ada/CAPS Compiler, Version 4.1

ACVC Version: 1.10

3.3 Ada Compiler

The Ada Compiler is invoked by running the program ADAC. The invocation command is described in Section 3.3.1.

Section 3.3.2 describes the input files needed by the compiler, and Section 3.3.3 explains what outputs the compiler produces. The program library is both an input and an output of the compiler, and is dealt with in Section 3.3.4.

3.3.1 Invocation Command

The command to invoke the Ada Compiler has the following syntax:

```
ADAC [ /Qualifier ... ] file-spec
```

Examples:

```
$ ADAC/LIST TESTPROG
$ ADAC/LIBRARY = MY_LIBRARY TEST -
```

Parameter:

o file-spec

The file-spec specifies the text file containing the source text of the compilation units to be compiled. If the file type is omitted in the source file specification, the file type .ADA is assumed by default. No wild card characters are allowed in the file specification.

If this parameter is omitted, the user will be prompted for it.

The allowed format of the source text is described in Section 3.3.2.2.

Qualifiers:

```
/CHECK (default)
/NOCHECK
```

This qualifier is used to suppress runtime checks from the generated object code. When the /NOCHECK qualifier is used, the effect is the same as if the unit(s) being compiled had included a pragma SUPPRESS for each of ACCESS_CHECK, DISCRIMINANT_CHECK, INDEX_CHECK, LENGTH_CHECK, RANGE_CHECK, DIVISION_CHECK, OVERFLOW_CHECK, ELABORATION_CHECK and STORAGE_CHECK. The inclusion of

pragma SUPPRESS in the unit being compiled takes precedence over the use of /CHECK as a compiler option. The result of /NOCHECK is a non-standard Ada program. The default is to include runtime checks in the object code.

```
/CMS GENERATIONS
/NOCMS_GENERATIONS (default)
```

This qualifier is used to pass DEC Code Management System (CMS) element generation information through the compiler system to the load module created from the Ada program. The CMS generation of each source file used in construction of the load module can be displayed using the CTRACE tool. See the Ada CAPS Object/Load Module Information Trace (User's Guide) for further information.

```
/CONFIGURATION_FILE = file-spec
/CONFIGURATION_FILE = ADACS_CONFIG (default)
```

This qualifier specifies the configuration file to be used by the compiler in the current compilation. If the qualifier is omitted, the configuration file designated by the logical name ADACS_CONFIG is used. Section 3.3.2.1 contains a description of the configuration file.

```
/DEBUG (default)
/NODEBUG
```

When /NODEBUG is active, the debugger file (having extension .DST) will not be created. The default is to create this file.

```
/KEEP = ( [ ASM, LST ] )
```

This qualifier is used to prevent the deletion of the .ASM and/or .LST files created by the macro assembler during compilation. The default is not to save either the .ASM or the .LST files. No corresponding /NOKEEP qualifier is allowed.

```
/LIBRARY = file-spec
/LIBRARY = ADACS_LIBRARY (default)
```

This qualifier specifies the current sublibrary and thereby also the current program library (cf. section 2).

If the qualifier is omitted the sublibrary designated by the logical name ADACS_LIBRARY is used as the current sublibrary (cf. section 2.1). Section 3.3.4 describes how the Ada compiler uses the library.

```
/LIST
/NOLIST                                (default)
```

When this qualifier is given, the source listing is written on the list file. Section 3.3.3.1 contains a description of the source listing.

If /NOLIST is active, no source listing is produced, regardless of any LIST pragmas in the program or any diagnostic messages produced.

```
/MACASM = ( [ CROSS_REFERENCE, STATISTICS, TIMING ] )
/MACASM = (NOCROSS_REFERENCE, NOSTATISTICS, NOTIMING)
                                         (default)
```

Inclusion of any of these parameters causes the macro assembler to include cross-reference information, statistical information, or timing information respectively in its generated list file. The source listing information is always included in the macro assembler list file. By default these sections are not included in the macro assembler list file.

Note: This qualifier is ignored unless the /KEEP = (LST) qualifier is also present.

```
/OBJECT                                (default)
/NOOBJECT
```

This qualifier indicates whether object code will be generated if the compilation is successful. Compilation is considerably faster when /NOOBJECT is specified, however the linker will report an error if an object file it requires is missing.

```
/OPTIMIZE [ = ( keyword [ , ... ] ) ]
/NOOPTIMIZE                            (default)
```

This qualifier specifies whether any optimizations should be performed on the generated code.

```
/OPTIMIZE
    Causes all optimizations to be performed.
```

```
/OPTIMIZE = CHECK
    The compiler will eliminate superfluous checks.
    This should NOT be used if /NOCHECK is used.
```

```
/OPTIMIZE = CSE
    Normal common subexpression elimination is
    performed.
```

/OPTIMIZE = PEEP

Peep hole optimization will be performed on the compiler generated code before object code, if any, is produced.

/OPTIMIZE = REORDERING

The compiler will try to reorder an aggregate with named component association into an aggregate with positional component association. It will also reorder named parameter association into an aggregate with positional parameter association.

/OPTIMIZE = STACK_HEIGHT

The use of temporary variables in expression evaluations is minimized.

/PRAGMA=UNIVERSAL_DATA

This qualifier is equivalent to placing the the statement, "pragma UNIVERSAL_DATA;" in the declarative part of the unit(s) being compiled. This allows the unit's volatile data to be located outside the first 64K of memory when the program is linked. See Appendix F of the Reference Manual for further information on this implementation-dependent pragma.

/PROGRESS

/NOPROGRESS (default)

When this qualifier is given, the compiler will output information about which pass the compiler is currently running. Default is not to output this information.

/SAVE_SOURCE

(default)

/NOSAVE_SOURCE

This qualifier specifies whether the source text is stored in the program library. If the source file contains several compilation units, the source text for each compilation unit is stored in the program library.

The source texts stored in the program library can be extracted using the PLU command TYPE.

/XREF

/NOXREF (default)

This qualifier is used to generate a cross-reference listing. If the /XREF qualifier is given and no severe or fatal errors are found during compilation, the cross-reference listing is written on the list file. The cross-reference listing is described in Section 3.3.3.1.